Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andrej Ota

Algoritem za iskanje najmanjšega poligona

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: prof. dr. Neža Mramor Kosta

Ljubljana, 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Cilj diplomske naloge je razviti in implementirati učinkovit algoritem za iskanje najmanjšega poligona, ki vsebuje dano točko in ima robove na podanih daljicah. Daljice se lahko med seboj poljubno sekajo, poligon pa je lahko konveksen ali konkaven, lahko pa ima tudi otoke. Algoritem naj bo časovno učinkovit, tako da bo uporaben tudi na podatkih, kjer je lahko daljic, na katerih ležijo robovi, relativno veliko.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Ota, z vpisno številko **24950395**, sem avtor diplomskega dela z naslovom:

Algoritem za iskanje najmanjšega poligona

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Neže Mramor Kosta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 1. julija 2014 Podpis avtorja:

Posvečeno Ivanki, Simonu in Mariji.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Iskanje presečišč med daljicami **5**

2.1 Vhodni podatki 5

2.2 Algoritem iskanja presečišč 6

2.3 Izhodni podatki 9

3 Iskanje najmanjših poligonov **11**

3.1 Vhodni podatki 11

3.2 Iskanje najmanjšega poligona 11

3.2.1 Priprava podatkov 12

3.2.2 Iskanje najmanjšega poligona 14

3.3 Normalizacija poligona 15

3.4 Izhodni podatki 17

4 Iskanje najmanjšega poligona in otokov **19**

4.1 Vhodni podatki 19

4.2 Iskanje najmanjšega poligona 19

4.3 Iskanje največjih otokov 20

4.4 Izhodni podatki 21

5 Implementacija **23**

5.1 Izbira razvojnega okolja 23

5.2 Natančnost računanja 24

5.2.1 Primer stabilnega algoritma 24

6 Sklepne ugotovitve **27**

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
CAD	computer-aided design	računalniško podprto načrtovanje
CCW	counter clock-wise	v nasprotni smeri urnega kazalca
CW	clock-wise	v smeri urnega kazalca

Povzetek

Določanje najmanjšega poligona z otoki je pogost problem v računalniško podprtem načrtovanju, kjer je treba, na primer, ugotavljati ploščine ali obode prosto skiciranih prečnih presekov ali načrtovati 3D objekte. Algoritem je največkrat dostopen kot funkcija zapolnitve površine v CAD aplikacijah. Podatki v teh aplikacijah so običajno podani kot seznam daljic, pri čemer vsa presečišča niso vnaprej znana. V diplomskem delu smo razvili algoritem za iskanje najmanjšega poligona z robovi na podani množici daljic, ki vsebuje dano izhodiščno točko, lahko pa ima tudi otoke. Algoritem je realiziran v več korakih. Vsak korak rešuje ločen geometrijski problem, rešitev pa predstavlja vhodne podatke za naslednji korak oziroma, pri zadnjem koraku, končni rezultat algoritma. Koraki so: izgradnja seznama daljic, ki se sekajo izključno v svojih krajiščih, določanje najmanjših poligonov, ki jih omejuje seznam daljic in iskanje najmanjšega poligona, ki vsebuje iskano točko, ter otokov v tem poligonu.

Ključne besede: računska geometrija, minimalni poligon.

Abstract

Minimal polygon search is a common problem in computer aided design when trying to determine surface of bounding areas on cross sections. An algorithm solving this problem is commonly implemented in CAD applications by a filling function. The input data are a set of line segments, where not all intersections are known. This thesis proposes an algorithm to find the minimal polygon with edges on segments from a given set which contains a given point of origin and which can also have holes. The algorithm reduces the problem into steps, where each step solves a single computational problem. The output data of each step become input data for the next step, or the end result of the algorithm from the final step. The steps of the algorithm are: a construction of line segment set where all line segments intersect only in their endpoints, finding minimal polygons defined by the set of line segments, and finding the minimal polygon containing the given point, including holes in that polygon.

Keywords: computational geometry, minimal polygon.

Poglavje 1

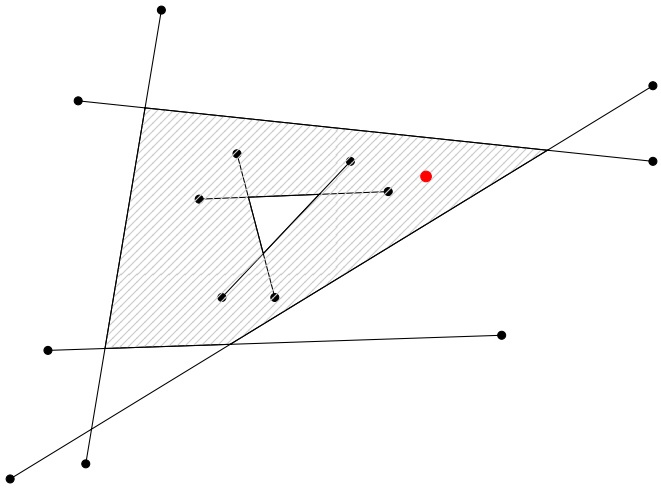
Uvod

Pri računalniško podprtem načrtovanju je pogosto potrebno iz seznama daljic, ki se lahko med seboj poljubno sekajo, izluščiti poligone, ki jih te daljice tvorijo kot osnovne površinske elemente, kot na primeru na sliki 1.1. Cilj diplomskega dela je razviti algoritem za iskanje minimalnega poligona, ki vsebuje dano točko in je določen s takšnim seznamom daljic. Dobljeni minimalni poligon je lahko konveksen ali konkaven, lahko pa ima tudi otoke.

Takšen poligon pogosto potrebujemo pri načrtovanju, za preproste operacije, kot na primer za izris zapolnitve območja (šrafiranje), izračun ploščine območja označenega s točko ali pa kot del bolj kompleksnih algoritmov, kot je izračun volumnov prostorskih oblik iz prečnih presekov. Algoritem, ki reši ta problem, je že implementiran v aplikacijah za računalniško podprto načrtovanje (CAD), različne rešitve problema pa najdemo tudi v strokovni literaturi [5] in na spletnih forumih [6].

Težava pri vseh teh pa je, da je govora o lastniških implementacijah, katerih koda ni dostopna, o idejah, ki niso bile do konca razdelane ali implementirane, ali pa so implementacije dostopne samo z licenco, ki je za potencialne uporabnike preveč omejujoča. Očitek implementaciji v aplikaciji AutoCAD® pa je tudi relativna počasnost, ki pride do izraza, kadar moramo postopek velikokrat ponoviti. Primer takšnega postopka, ki je občutljiv na hitrost iskanja minimalnega poligona, je izračun volumnov, kjer moramo najti

Slika 1.1: Rezultat iskanja poligona



minimalne poligone za veliko število prečnih presekov.

Možnih pristopov k reševanju problema je tako več, najbližji našem problemu je opisan v [5], vendar pa se konča pri določanju vseh minimalnih poligonov, ne določi pa minimalnega poligona, ki vsebuje iskano točko. Pri reševanju so uporabili podoben osnoven pristop in algoritem kot celoto razdelili na manjše korake, ki se lahko izvedejo z že znanimi elementarnimi algoritmi. Pri tem smo uporabili na vsakem koraku algoritem, ki vrne izhodne podatke v obliki uporabni za vhodne podatke na naslednjem koraku. Rezultat zadnjega elementarnega algoritma oziroma koraka pa vrne minimalni poligon in seznam maksimalnih otokov, če ti obstajajo.

Elementarni algoritmi, iz katerih smo sestavili celoten algoritem so tako sledeči:

- Za vhodne podatke, ki jih predstavlja seznam daljic AB v ravnini, podanih v obliki, kjer sta A in B krajišči, s pomočjo algoritma za iskanje presečišč med daljicami določimo vsa presečišča in daljice razdelimo tako, da se sekajo samo še v svojih krajiščih. Postopek je podrobneje opisan v poglavju 2.

- Iz daljic, ki se stikajo samo še v svojih krajiščih, sestavimo sklenjene

lomljene črte, ki predstavljajo obode minimalnih in maksimalnih poligonov, kot smo opisali v poglavju 3.

- Iz seznama poligonov, ki so podani kot sklenjene lomljene črte, poiščemo minimalni poligon, ki vsebuje iskano točko, in za ta poligon tudi vse maksimalne poligone, ki so v njegovi notranjosti. Postopek je opisan v poglavju 4, elementarna operacija v tem koraku pa je testiranje, če se točka nahaja znotraj poligona.

Razdelitev na posamične korake pa je dobrodošla tudi iz izvedbenega vidika, saj se lahko z novimi spoznanji, z dodatnimi zahtevami ali pa že samo z željo izboljšanja celotnega algoritma enostavno popravi (ali pa v celoti zamenja) samo en korak. Pri tem preostali koraki ostanejo nespremenjeni, če ostanejo enake tudi lastnosti vmesnih podatkov. To pomeni, da ima programska koda, ki implementira algoritem, tudi lastnosti, ki omogočajo testiranje posamičnih korakov (ang. "profiling") neodvisno od celote, kot celoto pa se jo tudi lažje vzdržuje in odpravlja morebitne napake v implementaciji algoritma.

Poglavje 2

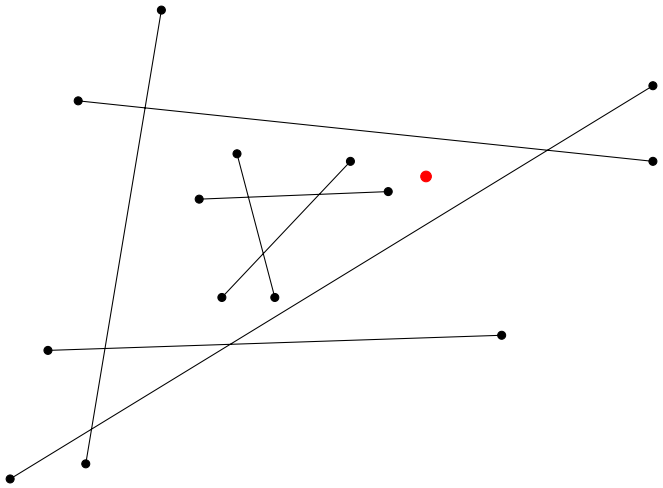
Iskanje presečišč med daljicami

2.1 Vhodni podatki

Prvi korak algoritma prevede seznam daljic, ki se lahko med seboj poljubno sekajo, tako kot na sliki 2.1, v seznam daljic, ki se sekajo samo v svojih krajiščih. Da to dosežemo, pa moramo poiskati vsa presečišča med pari daljic. Naiven pristop, ki preverja obstoj presečišča med vsemi obstoječimi pari, ima časovno zahtevnost $O(n^2) = \Theta(n^2)$, kjer je n število daljic. To pomeni, da bo ne glede na optimizacijo vsakega posamičnega koraka pri podvojitvi števila daljic čas izvajanja početverjen.

Pri implementaciji se bomo omejili na primer podatkov, kjer se nobeni dve daljici med seboj delno ali v celoti ne prekrivata, saj se pri podatkih v praktični uporabi daljice zelo redko prekrivajo. Še več, velikokrat se takšen primer razume kot napaka v podatkih, ki jo je potrebno odpraviti ne glede na omejitve našega algoritma. Ta omejitev tudi ni huda, saj je možno algoritem v tem koraku dopolniti tako, da bo pravilno obravnaval tudi takšne primere, vendar pa se bomo temu delu izognili zato, da prihranimo pri času razvoja in pri kompleksnosti implementacije.

Slika 2.1: Vhodni podatki



2.2 Algoritem iskanja presečišč

Algoritem, ki je časovno zahtevnost prvi zmanjšal pod mejo$O(n^2)$, je prečesevalni algoritem (ang. "plane sweep algorithm") Bentelya in Ottmanna [2], kjer se iskanje presečišča omeji samo na tiste pare daljic, kjer obstaja možnost, da se sekajo. S tem algoritmom se časovna zahtevnost iskanja zniža na $O((n + k) \log n)$, kjer je n število daljic, k pa število presečišč. To je pa bistveno izboljšanje glede na naiven algoritem, ki ima zaradi izčrpnega preiskovanja časovno zahtevnost $O(n^2)$.

Pri algoritmu je bistveno vprašanje, kako se izogniti iskanju presečišča med dvema daljicama, ki sta daleč narazen. Najlažje to storimo tako, da najprej preverimo, če se projekciji daljic na abscisno os prekrivata, kajti če se ne, potem se tudi daljici ne moreta sekati. Zatorej moramo primerjati samo tiste daljice, katerih projekcije se prekrivajo. Da bomo našli ta prekrivanja, si lahko predstavljamo navpično premico, s katero prečešemo ravnino v smeri od manjših x koordinat proti večjim, začenši s koordinato, ki je manjša od najmanjše x koordinate v vhodnih podatkih.

Ko premico premikamo proti desni, lahko v vsakem položaju ugotovimo, katere daljice se s to premico sekajo in jih uvrstimo na seznam. Na tem se-

znamu so kandidati za pare daljic, ki se med seboj sekajo. Če za ta "status" uporabimo urejen seznam S, kjer so daljice urejene po naraščajoči y koordinati presečišča, ugotovimo, da se vsebina S spremeni samo v krajiščih (dodamo ali odvzamemo daljico) ali presečiščih (daljici zamenjata vrstni red). Algoritem mora tako spremeniti "status" in preveriti obstoj morebitnih presečišč samo v teh kritičnih točkah. Konkretno to pomeni:

- ko se v seznam S na položaj i doda nova daljica, se preveri obstoj presečišča z daljicami na novih položajih $i-1$ in $i+1$;
- ko se v seznam S iz položaja i odstrani daljica, se preveri obstoj presečišča z daljicami na novih položajih $i-1$ in i;
- ko se v seznamu S zamenjata daljici na položajih i in $i+1$, se preveri obstoj presečišča z daljicami na novih položajih, in sicer paroma $i-1$ in i ter $i+1$ in $i+2$.

Implementacija algoritma iskanja presečišč uporablja prednostno vrsto točk Q, ki je urejena po naraščajoči x koordinati točke, na začetku postopka pa v njo vnesemo vsako krajišče iz vhodnih podatkov. Tekom postopka v Q tudi dodamo vsako najdeno krajišče. Ko bomo iz vrste odstranjevali točke v vrstnem redu od najmanjše proti največji in premikali prečesevalno premico na vsako novo x koordinato odstranjene točke, bomo sočasno vzdrževali urejen seznam daljic S, kjer bodo te daljice urejene po y koordinati presečišča s trenutnim položajem prečesevalne premice. Algoritem z delom zaključi, ko v prednostni vrsti Q več ni točk za obravnavo. S tem smo izvedli prečesevalni algoritem, kot je bil opisan v [2].

Algoritem v opisani osnovni izvedbi deluje samo za podatke, ki zadoščajo naslednjim pogojem:

1. nobeno krajišče ali presečišče si ne deli koordinate x z nobenim drugim krajiščem ali presečiščem;
2. skozi vsako presečišče gresta največ dve daljici;

3. nobeni dve daljici se ne prekrivata.

Osnovni algoritem smo dopolnili tako, da deluje tudi na podatkih, ki ne izpolnjujejo prvega in drugega od zgoraj navedenih pogojev. Različna krajišča ali presečišča daljic imajo lahko torej tudi enako koordinato in skozi eno točko gre lahko tudi več daljic. Edina izjema, ki je algoritem ne obravnava, je torej prekrivanje daljic.

Prvo izjemo odpravimo tako, da razširimo pravilo za urejanje točk v vrsti Q tako, da predpišemo strogo linearno urejenost z operatorjem $<$ s predpisom (gl. [3], 2. poglavje, str. 24):

$$A < B \iff x_A < x_B \vee (x_A = x_B \wedge y_A < y_B)$$

Spremenjeno pravilo omogoči, da lahko v vrsto Q vstavimo tudi točke (krajišča ali presečišča), ki si delijo x koordinato. S tem je izpolnjen tudi predlog v opisu osnovnega algoritma, da lahko za odpravo omejitve točke ustrezno uredi še pred glavno zanko, čeprav implementacije predloga avtorja nista natančneje opisala.

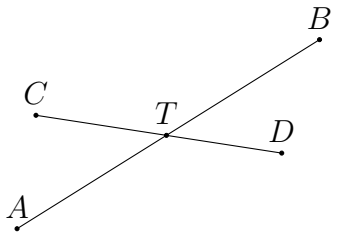
Druga omejitev je bila odpravljena tako, da vsako presečišče T že med samim postopkom spremenimo v krajišče tako, da se daljici AB in CD, ki se sekata v T, iz podatkov odstranita, namesto njiju pa se vstavi nove daljice AT, TB, CT in TD. Ko bo algoritem obravnaval presečišče T, bo ta obravnavana enako kot druga krajišča, kjer bo iz seznama S odstranil daljice, ki se, gledano iz leve proti desni, v tej točki končajo, ter dodal daljice, ki se v tej točki začno. S tem se izognemo tudi dodatni izjemi več daljic, ki se v tej točki sekajo.

Za primer podatkov na sliki 2.2 je podana tabela stanj algoritma na tabeli 2.1. Iz tabele je vidna razlika na prehodu med korakoma C in T, kjer mora originalen algoritem spremeniti vrstni red daljic v S, medtem ko novi algoritem samo doda daljice, ki imajo "najmanjšo" točko v T, potem ko je odstranil daljice, ki so se v tej točki končale.

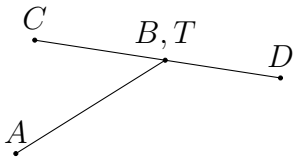
Delovanje algoritma v delu, ki obravnava trenutno točko iz prednostne vrste Q, poenostavljeno opisuje psevdokoda 2.1. Implementacija obravnava

Slika 2.2: Delitev daljic

(a) Običajna delitev daljic



(b) Poseben primer



tudi posebne primere, kjer je T enaka enemu izmed krajišč, zaradi česar rezultat niso štiri, temveč samo tri nove daljice (gl. sliko 2.2b), vendar teh pogojev v psevdokodo nismo posebej vključili, saj so dovolj enostavni, da jih lahko vsakdo doda sam, hkrati pa bi zakrili bistvo spremembe prvotnega algoritma.

Edini izračun, ki name še ostane, je določanje presečišča med dvema daljicama – sosedama v seznamu S – ki ga med daljicama AB in CD določimo po naslednjem postopku:

1. V enačbo nosilne premice daljice AB vstavimo točki B in C. Če je produkt teh rezultatov pozitivno število, potem presečišče ne obstaja.

2. V enačbo nosilne premice daljice CD vstavimo točki A in B. Če je produkt teh rezultatov pozitivno število, potem presečišče ne obstaja.

3. Izračunamo rešitev sistema dveh linearnih enačb z dvema spremenljivkama x in y, ki predstavljata koordinati presečišča T.

2.3 Izhodni podatki

Na koncu koraka se vse daljice sekajo samo in samo v izhodiščih. S tem je izpolnjena zahteva za vhodne podatke v naslednjem koraku.

Tabela 2.1: Tabela stanj

korak	Q	S
	$\{A, C, D, B\}$	$\emptyset$
A	$\{C, D, B\}$	$\{AB\}$
C	$\{T, D, B\}$	$\{AB, CD\}$
T	$\{D, B\}$	$\{CD, AB\}$
D	$\{B\}$	$\{AB\}$
B	$\emptyset$	$\emptyset$

prvotni algoritem

korak	Q	S
	$\{A, C, D, B\}$	$\emptyset$
A	$\{C, D, B\}$	$\{AB\}$
C	$\{T, D, B\}$	$\{AT, CT\}$
T	$\{D, B\}$	$\{TD, TB\}$
D	$\{B\}$	$\{TB\}$
B	$\emptyset$	$\emptyset$

popravljen algoritem

segments: seznam daljic, ki se začno v točki v trenutnem koraku

```
begin
    foreach s in segments:
    do
        insert s into S;
        lower ← previous_neighbour of s in S;
        if T = lower ∩ s; T ≠ ∅ then
            /* Odstrani lower iz S in namesto nje dodaj novo
               daljico, ki se konča v T                    */
            remove lower from S;
            add Segment(lower left, T) to S;
            /* Odstrani s iz S in namesto nje dodaj novo
               daljico, ki se konča v T                    */
            remove s from S;
            add Segment(s left, T) to S;
            s ← (s left, T);
        end
    end
end
```

Psevdokoda 2.1: Obravnava točke

Poglavje 3

Iskanje najmanjših poligonov

3.1 Vhodni podatki

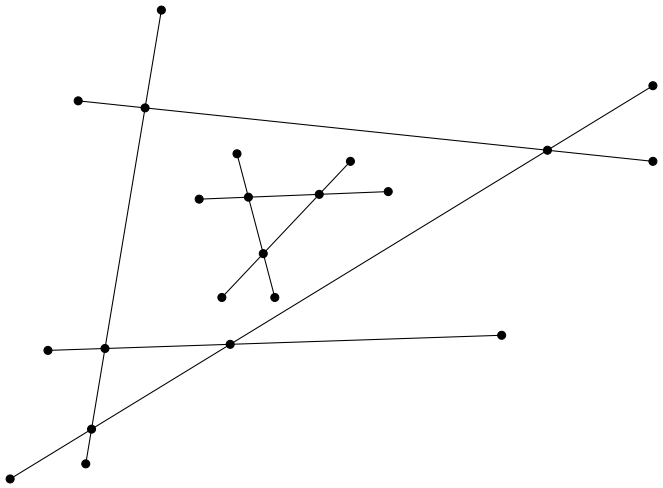
Vhodni podatek za algoritem za konstrukcijo najmanjših poligonov je seznam daljic, ki se sekajo samo in samo v svojih krajiščih (gl. sliko 3.1). Iskanje najmanjših poligonov prevedemo na problem iskanja seznama povezav med točkami, ki opišejo najkrajšo pot z začetkom in koncem v isti točki.

3.2 Iskanje najmanjšega poligona

Algoritem za iskanje vseh najmanjših poligonov je opisan v [5], vendar pa ima predlagana rešitev časovno zahtevnost $O(n^4)$, kjer je n število vseh krajišč. To omejuje njeno uporabnost na podatkih, ki vsebujejo že nekaj tisoč krajišč. Ocenili smo, da je osnovna omejitev predhodnega algoritma v uporabi Dijkstrovega algoritma za iskanje najkrajše povezave, ki za vsako izhodišče vedno znova obišče vse točke na grafu. To pomeni, da je geometrični problem preveden zgolj na problem iskanja najmanjših ciklov v grafu, s tem pa so izpuščene bolj optimalne rešitve, ki upoštevajo geometrijo problema, ki je graf ne more zajeti.

Kot boljšo rešitev predlagamo algoritem, ki je podoben algoritmu za določanje presekov dveh seznamov poligonov, opisanem v [3], 2. pogl., str.

Slika 3.1: Vhodni podatki

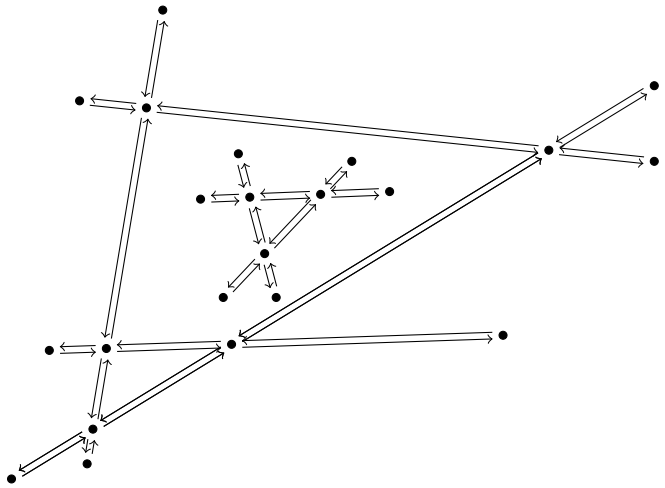


29–39, in vključuje še algoritem za določanje smeri, v kateri so naštete točke oboda poligona (CW ali CCW), kot ga je opisal Alciatore v [1]. Na ta način smo Dijkstrov algoritem nadomestili z algoritmom, ki uporablja prednostno vrsto (angl. "priority queue") in ima manjšo časovno zahtevnost. Kljub temu pa ta časovna zahtevnost ni majhna, saj moramo še vedno uporabiti tri urejene sezname oziroma prednostne vrste, ki pa posamično doprinesejo časovno zahtevnost reda $O(\log n)$, če jih implementiramo z enostavnim binarnim iskalnim drevesom. Časovna zahtevnost je tako reda $O(n^3 \log n)$, kar je ugodneje od predhodne $O(n^4)$ zahtevnosti.

3.2.1 Priprava podatkov

Pred začetkom iskanja moramo podatke ustrezno pripraviti, kar storimo tako, da vsa krajišča daljic vstavimo v prednostno vrsto. Ta je po urejenosti in namenu enaka prednostni vrsti, ki jo uporablja prečesevalni algoritem. Hkrati pa zgradimo tudi usmerjen graf tako, da vsaki daljici priredimo dve povezavi, v vsako smer eno. Za daljico AB tako ustvarimo povezavi $\vec{AB}$ in $\vec{BA}$. Vsakega izmed tako dobljenih vektorjev vstavimo v urejen seznam vezan na točko – vozlišče, kjer se vektor začne. S pomočjo dobljenega grafa bomo iskali

Slika 3.2: Pripravljeni podatki pred začetkom iskanja



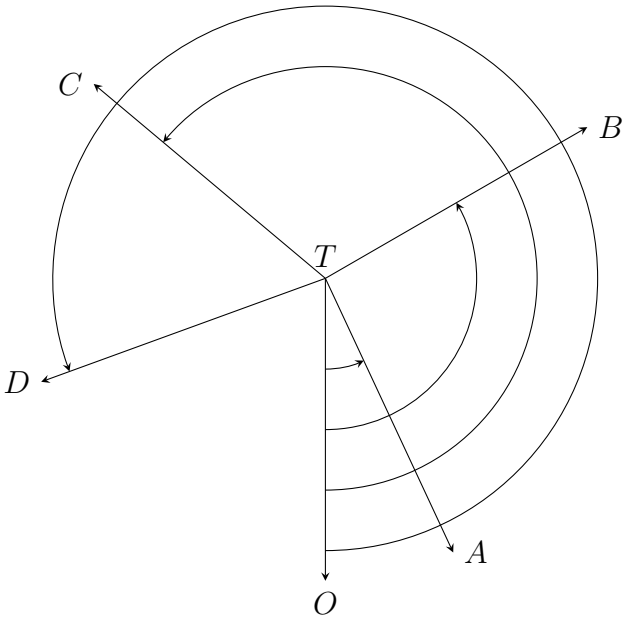
najmanjši cikel na način, ki upošteva tudi geometrične lastnosti podatkov, ne pa zgolj razdalj med točkami. Primer tako pripravljenih podatkov je na sliki 3.2.

Seznami, ki smo jih zgradili, so urejeni na sledeče načine:

- **Seznam vseh krajišč daljic**
  Točke so urejeno leksikografsko od najmanjše (z najvišjo prednostjo) do največje. Urejenost je podana z enakim pravilom kot v prvem koraku algoritma (gl. poglavje 2).

- **Seznam vseh povezav iz točke**
  Povezave so urejene z naraščajočim kotom, merjenim v nasprotni smeri urinega kazalca med vektorjem z začetkom v izhodiščni točki, ki kaže navzdol, in vektorjem, ki je določen s povezavo. Kot je omejen na interval $[0, 2\pi)$. Primer urejenosti vektorjev je prikazan na sliki 3.3, kjer so vektorji, urejeni od prvega do zadnjega: $\vec{TA}$, $\vec{TB}$, $\vec{TC}$, $\vec{TD}$.

Slika 3.3: Urejenost vektorjev



3.2.2 Iskanje najmanjšega poligona

Ko so podatki pripravljeni, najmanjši poligon iščemo z vgnezdenimi zankami, ki dostopajo do podatkov v urejenih seznamih, in jih po končani obravnavi iz teh seznamov tudi brišejo. Iz prednostne vrste točk se po vrsti prevzema točke, ki predstavljajo začetno točko poligona. Dokler obstajajo povezave iz izbrane točke, ponavljamo postopek iskanja oboda v smeri urinega kazalca (CW), s katerim najdemo bodisi minimalni poligon v CW smeri, ali pa maksimalni poligon, če ugotovimo, da je bila smer nasprotna smeri urinega kazalca (CCW). Ta postopek, ki predstavlja zunanjo zanko algoritma, zaključimo, ko prednostno vrsto točk izpraznimo.

Postopek iskanja CW oboda začnemo s tisto povezavo iz točke, ki ima najmanjši kot (torej je prva na urejenem seznamu povezav iz trenutne točke). Ko to določimo, se prestavimo iz začetne točke v naslednjo, ki je na koncu najdenega vektorja, povezavo pa izbrišemo iz seznama. V vseh naslednjih točkah nato postopek ponavljamo s to razliko, da izberemo povezavo, ki ima

najmanjši kot, ki je večji od kota povezave, preko katere smo prišli v trenutno točko, gledano iz te točke.

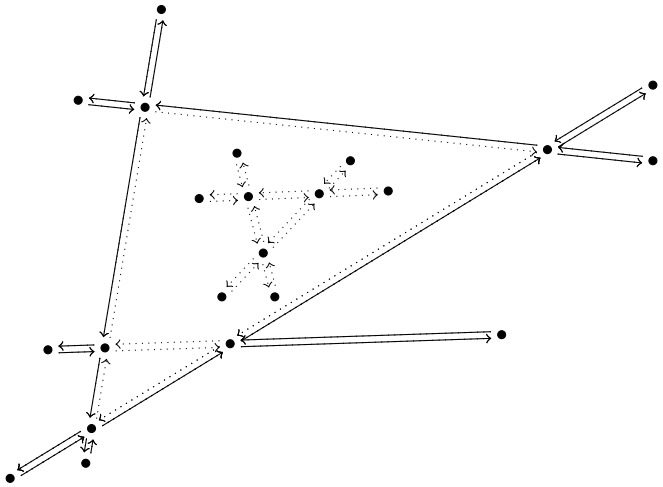
Če smo v točko B prišli preko povezave $\vec{AB}$, potem najdemo tisto povezavo $\vec{BX}$, ki ima najmanjši kot, ki je še vedno večji od kota $\vec{BA}$. Postopek zaključimo, ko se vrnemo v izhodiščno točko. S tem smo opisali bodisi maksimalni poligon, kot je prikazano na sliki 3.4, bodisi minimalni poligon, kot je prikazano na sliki 3.5.

Rezultat tega algoritma pa niso samo najmanjši poligoni, temveč tudi poligoni, ki opišejo zunanji rob vsakega grafa izgrajenega iz vhodnih podatkov. Ti poligoni se razlikujejo od najmanjših poligonov v tem, da seznam točk oboda ne opiše v smeri urinega kazalca, kot to velja za najmanjše poligone, temveč opišejo pot v nasprotni smeri (CCW). V podatkih, ki vsebujejo vsaj en poligon, bo vedno obstajal vsaj en takšen poligon (gl. sliko 3.5). Lahko pa je takšnih poligonov tudi več, saj je njihov obstoj vezan na posamične grafe, ki se jih ustvari pri pripravi podatkov. Teh grafov pa je lahko tudi več, saj ni nujno, da vhodni podatki predstavljajo en sam povezan graf. Ravno nasprotno: nepovezani grafi se pričakujejo, saj so ravno otoki primer takšnih dodatnih grafov (niso pa vsi takšni nepovezani grafi tudi res otoki). To pa je tudi dodatna pozitivna lastnost uporabljenega algoritma, saj bo ravno podatek o vrstnem redu točk, ki opisujejo poligon, bistveno prispeval k enostavnosti naslednjega koraka.

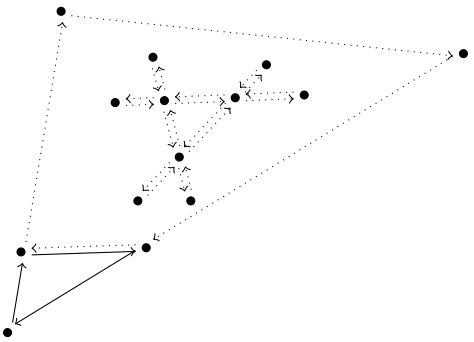
3.3 Normalizacija poligona

Poligoni, najdeni z opisanim algoritmom, lahko vsebujejo tudi degenerirane dele z ničelno ploščino. Takšni degenerirani izrastki so moteči, zato jih odstranimo tako, da se sprehodimo skozi vsa oglišča poligona in v vsaki točki preverimo, ali se točka A v leksikografski ureditvi ujema s točko C dva koraka naprej. Če se, je vmesna točka B odveč, saj je trikotnik ABC degeneriran. Točki B in C v tem primeru odstranimo iz seznama oglišč poligona. Ko ta postopek ponovimo za vse točke na obodu, bo poligon normaliziran.

Slika 3.4: Prvi najden poligon – obodni poligon, CCW



Slika 3.5: Prvi najden najmanjši poligon – CW poligon



Kot zadnji korak za vsak najdeni poligon z uporabo Gaussove formule za izračun ploščine poligona (Beyer, str. 123–124 [4]) ugotovimo, ali je orientacija, ki jo določa zaporedje njegovih oglišč, v smeri urinega kazalca (CW) ali v nasprotni smeri (CCW). Ta podatek bo prišel prav v naslednjem koraku, kjer določimo, ali je iskana točka v notranjosti ali zunanjosti najdenega minimalnega poligona, saj nam smer pove, če je govora o minimalnem poligonu (ta je CW) ali maksimalnem poligonu (ta je CCW). To preverimo s ponovnim sprehodom skozi oglišča poligona in računanjem njegove površine z Gaussovo metodo. Ta nam da pozitivno vrednost, kadar je poligon naveden v CW, sicer pa negativno vrednost.

3.4 Izhodni podatki

Izhodni podatki tega koraka vsebujejo vse minimalne in vse maksimalne poligone. Minimalni poligoni so tisti, ki ne vsebujejo nobenega drugega povezanega poligona, lahko pa vsebujejo nepovezane poligone, ki predstavljajo otoke. Njihova lastnost je, da imajo oglišča navedena v smeri urinega kazalca. Maksimalni otoki so tisti, ki opišejo največji obod posameznega grafa, njihova lastnost pa je, da so oglišča navedena v smeri, ki je nasprotna smeri urinega kazalca.

Čeprav je bil primarni cilj tega koraka iskanje vseh minimalnih poligonov, saj moramo kot rezultat vrniti tistega, ki vsebuje iskano točko, pa bomo obdržali tudi vse maksimalne poligone, saj bomo z njimi lahko določili morebitne otoke.

Poglavje 4

Iskanje najmanjšega poligona in otokov

4.1 Vhodni podatki

Vhodni podatki vsebujejo poleg zaporedja oglišč poligona tudi njegovo orientacijo, ki je podana s smerjo (CW or CCW), v kateri si oglišča sledijo. Ta dodaten podatek nam pomaga, da v tem koraku ločimo minimalne (ti so CW) in maksimalne poligone (ti so CCW), ki jih uporabljamo na različen način. Iz nabora prvih določimo najmanjši poligon, ki vsebuje iskano točko, iz nabora slednjih pa vse največje otoke, ki so vsebovani v predhodno najdenem najmanjšem poligonu.

4.2 Iskanje najmanjšega poligona

Najmanjši poligon, ki vsebuje dano točko, poiščemo s sprehodom čez vse CW orientirane poligone. Na vsakem koraku preverimo, ali novi poligon vsebuje dano točko in ali je manjši od trenutno najmanjšega najdenega poligona. Če sta oba pogoja izpolnjena, poligon shranimo kot trenutno najmanjšega in ponovimo korak z naslednjim poligonom iz seznama. Ob koncu postopka smo našli poligon, ki dano točko vsebuje in hkrati ne vsebuje nobenega drugega

poligona iz seznama CW orientiranih poligonov.

Iskanje smo izvedli z grobo silo, ki ima časovno zahtevnost razreda $O(n)$, kjer je n število poligonov. Pri tem za vsak poligon testiramo, če vsebuje iskano točko in če je katera koli njegova točka vsebovana znotraj že predhodno najdenega kandidata.

4.3 Iskanje največjih otokov

Postopek iskanja največjih otokov je računsko nekoliko bolj potraten, pomaga pa to, da lahko v tipičnih podatkih, kot so bili testni podatki iz domene gradbeništva, večino poligonov izločimo že s primerjanjem najmanjših in največjih koordinat. Pri tem zmanjšanju potratnosti posamičnega koraka je bil odločilen dejavnik to, da je delež konkavnih poligonov majhen, tisti pa, ki so takšni, v konkavnem delu ne vsebujejo veliko poligonov, za katere bi nato bilo potrebno preverjati, če so otoki ali ne. V kolikor bi se srečali s podatki, ki bi bistveno odstopali od te običajne oblike, bi se hitro pokazala časovna potratnost v tem koraku, saj je časovna zahtevnost $O(n^2)$, kjer je n število poligonov.

Druge poenostavitve in lastnosti podatkov, ki pospešujejo preverjanje pa so:

1. Preverjajo se samo CCW poligoni – teh je največ toliko kolikor je CW poligonov, tipično pa jih je veliko manj.

2. Pri testiranju, ali leži en poligon znotraj drugega, se uporablja prva točka poligona, ki ga preverjamo. Zaradi oblike podatkov, ki je rezultat predhodnih korakov, se ne more primeriti, da se bi dva poligona sekala v kateri koli točki. Če obravnavamo poligona P in R, potem velja, da če je katero koli oglišče $p \in P$ znotraj R, potem so vsa oglišča p znotraj R. Hkrati pa velja tudi, da če oglišče $p \in P$ ni znotraj R, potem ni nobeno oglišče p znotraj R.

3. Pri testiranju upoštevamo, da se dva CCW poligona ne moreta dotikati, kar tudi izhaja iz oblike podatkov.

4. Pri testiranju upoštevamo, da je točka na obodu poligona vsebovana v poligonu.

Končen rezultat tega algoritma je seznam vseh največjih otokov znotraj najmanjšega poligona, ki vsebuje iskano točko.

4.4 Izhodni podatki

Izhodni podatki na koncu tega koraka so tudi izhodni podatki celotnega algoritma. Rezultat je tako sestavljen iz CW orientiranega najmanjšega poligona, ki vsebuje iskano točko, in seznama, ki vsebuje same največje CCW orientirane poligone, ki so otoki znotraj najmanjšega poligona. S tem je celoten algoritem zaključen.

Poglavje 5

Implementacija

5.1 Izbira razvojnega okolja

Uporabnost kode je omejena z možnostjo njene uporabe za konkretne aplikacije. Pri izbiri okolja smo tehtali tako med enostavnostjo implementacije, kot tudi neposredno uporabnostjo te implementacije v čim več aplikacijah. Izbrali smo C++, in sicer dialekt C++98 s STL. Razlogov za to izbiro je bilo več, najpomembnejši pa so bili:

1. Možnost uporabe osnovnih gradnikov, kot sta razreda std::list in še zlasti std::map, s čemer smo se izognili potrebi po lastni implementaciji binarnih iskalnih dreves uporabljenih za prednostne vrste in urejene sezname.

2. Implementacija std::map je v večini STL knjižnic in tudi standardnih C++ knjižnic (od C++11 naprej) rdeče-črno drevo, ki ima zahtevano časovno zahtevnost $O(\log n)$ za vstavljanje in iskanje.

3. C++98 standard je še vedno najbolj razširjen, hkrati pa so v kodi uporabljene funkcije tiste, ki jih vključuje tudi trenutno aktualen C++11 standard.

4. C++ kot programski jezik, s katerim se ustvarjajo knjižnice, je tako uporaben in popularen, da je možno takšno knjižnico neposredno po-

vezati s celo vrsto drugih razvojnih okvirjev. Java in Python sta primer dveh zelo popularnih razvojnih ogrodij, ki imata možnost vključevanja zunanjih knjižnic napisanih v C++.

5.2 Natančnost računanja

Pri računanju s plavajočo vejico se vedno pojavi problem izgube decimalnih mest (angl. "truncation"). Do izgube natančnosti pride povsod tam, kjer dejanskega števila ni možno natančno predstaviti z razpoložljivim številom decimalnih mest. Pri vsakem naslednjem izračunu s takšno vrednostjo pa se napaka še povečuje. Da smo algoritme stabilizirali oziroma preprečili odstopanja, ki so posledica zaokrožitvenih napak, je bilo potrebno za konkretne podatke ugotoviti, kakšno natančnost zahtevajo. Na podlagi tega pa nato izbrati primeren ε, ki bo določal največjo razliko med številoma, pri kateri se bosta ti šteli za enaki. Koda ima za ta namen uporabljeno globalno spremenljivko in posebno funkcijo, ki ugotavlja, če je parameter znotraj $[-\varepsilon, \varepsilon]$ ali ne.

Na takšna odstopanja je še zlasti občutljiv algoritem iskanja presečišč, kjer se lahko zgodi, da zaradi zaokrožitvene napake izračunano presečišče ne leži natančno na daljici.

5.2.1 Primer stabilnega algoritma

Kot primer stabilnega algoritma, ki poskuša čim bolj omejiti vpliv te napake in je zato dejansko drugačen, kot bi lahko bil, če bi imeli možnost neskončne natančnosti ali pa računanja s pravimi racionalnimi števili, je funkcija urejenosti za seznam povezav iz točke, kjer morajo biti točke urejene glede na kot. Algoritem ne izračunava dejanskega kota, ker bi to pomenilo uporabo trigonometričnih funkcij, temveč za urejanje uporablja smerni koeficient premice, ki je nosilka vektorja, in podatek o kvadrantu, v katerem se ta vektor nahaja. Ravno tako pa ne uporablja neposredno smernega koeficienta, temveč v odvisnosti od kvadranta uporablja bodisi smerni koeficient, bodisi

njegovo obratno vrednost. S tem ohrani maksimalno natančnost, ko se ta koeficient približuje navpični smeri (koeficient bi šel proti neskončnosti) ali vodoravni smeri (obratna vrednost koeficienta bi šla proti neskončnosti), kar bi brez upoštevanja kvadranta privedlo do nagle in nepotrebne izgube pri natančnosti.

Poglavje 6

Sklepne ugotovitve

Algoritem smo implementirali za potrebe podjetja, ki ga želi uporabiti v lastnem produktu. Primerjalno je implementacija algoritma za testne podatke hitrejša od implementacije, ki jo trenutno uporablja AutoDesk AutoCAD®. S tem smo tudi izpolnili pričakovanje podjetja, da se ta računski postopek pospeši.

Algoritem ima še vedno omejitev pri vhodnih podatkih (daljice se ne smejo prekrivati – poglavje 2), vendar pa se lahko tudi ta omejitev odpravi, če se bo za to v prihodnosti pokazala potreba. Tudi če bi se odpravilo omejitev, da vhodni podatki ne smejo vsebovati poligonov, ki se prekrivajo, se sama časovna zahtevnost algoritma ne bi spremenila. Ravno časovna zahtevnost pa je tista, ki pri povečanju količine vhodnih podatkov bistveno vpliva na dejanske čase izvajanja.

Vsak posamičen korak bi se verjetno dalo še dodatno optimizirati, vendar pa bi to objektivno povečalo obseg kode, koristi pa bi bile vprašljive. V praktični uporabi se ne najdejo podatki, ki bi predstavljali primere na ali blizu zgornje meje časovne zahtevnosti, sama koda pa bi se povečala in s tem postala tudi potencialno večje breme za vzdrževanje.

Literatura

[1] D. G. Alciatore, R. Miranda. "A Winding Number and Point-in-Polygon Algorithm", Department of Mechanical Engineering Colorado State University, Fort Collins, CO, 1995.

[2] J. L. Bentley, T. A. Ottmann. "Algorithms for reporting and counting geometric intersections", *IEEE Transactions on Computers*, št. C-31, zv. 9, str. 643–647, 1979.

[3] M. de Berg et al. *Computational Geometry: Algorithms and Applications (3rd ed.)*. Springer-Verlag, 2008.

[4] W. H. Beyer (ured.) *CRC Standard Mathematical Tables, 28th ed.*. CRC Press, 1987.

[5] A. Ferreira et al. "Polygon Detection from a Set of Lines", *Actas do 12º Encontro Português de Computação Gráfica (12th EPCG)*, str. 159–162, 2003.

[6] stackoverflow.com (2014) "How to find the polygon enclosing a point from a set of lines" Dostopno na: http://stackoverflow.com/questions/16140831/how-to-find-the-polygon-enclosing-a-point-from-a-set-of-lines